AFRL-VS-HA-TR-98-0061

AN APPROACH TO THE MODERNIZATION OF THE GOVERNMENT REFERENCE PHENOMENOLOGY CODES

W. K. Cobb
M. Lowrey
R. R. Gamache

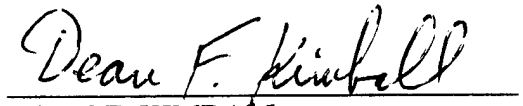
University of Massachusetts Lowell
Department of Environmental, Earth, and Atmospheric Sciences
1 University Avenue
Lowell, MA 01854

21 May 1998

Scientific Report No. 1

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
29 Randolph Road
AIR FORCE MATERIEL COMMAND
HANSCOM AFB, MA 01731-3010

This technical report has been reviewed and is approved for publication.



DEAN F. KIMBALL
Contract Manager
AFRL/VSSS

DUANE E. PAULSEN
Deputy Chief
AFRL/VSSS

This report has been reviewed by the ESC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS).

Qualified requestors may obtain additional copies from the Defense Technical Information Center (DTIC). All others should apply to the National Technical Information Service (NTIS).

If your address has changed, if you which to be removed from the mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/VSIM, 29 Randolph Road, Hanscom AFB MA 01731-3010. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	21 May 1998	Scientific Report No. 1

4. TITLE AND SUBTITLE
An Approach to the Modernization of the Government Reference Phenomenology Code

5. FUNDING NUMBERS
PE: 63872F
PR: S321
TA: GE
WU: FC

Contract #: F19628-96-C-0142

6. AUTHOR(S)
W. K. Cobb, M. Lowrey, R.R. Gamache

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Massachusetts/Lowell
Department of Environmental, Earth and Atmospheric Sciences
1 University Avenue
Lowell, MA 01854

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
29 Randolph Road
Hanscom AFB MA 01731-3010

Contract Manager: Dean Kimball/VSBM

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-VS-HA-TR-98-0061

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many of the reference phenomenology modeling codes in use today have grown out of codes and fragments of codes originally written more than 20 years ago - before modern ideas of software engineering had been developed and disseminated. While the codes are computationally efficient, they have significant correctable structural and architectural shortcomings. These shortcomings make maintenance and enhancement of the codes difficult and costly, and greatly complicate verification, validation, and optimization of the codes.

In this report, we consider the process of modernizing reference phenomenology modeling odes using modern software methodologies and language features. The ultimate aim of this work is to propose a method for updating these codes, which improves the speed, reliability, maintainability, verification an validation of the codes. We also address the creation of platform independent codes that can run on any computer system with proper language compiler.

14. SUBJECT TERMS
Computer code modernization, Modern computer languages, Modular programming, FOTRAN, C/C++

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Many of the reference phenomenology modeling codes in use today have grown out of codes and fragments of codes originally written more than 20 years ago – before modern ideas of software engineering had been developed and disseminated. While the codes are computationally efficient, they have significant correctable structural and architectural shortcomings. These shortcomings make maintenance and enhancement of the codes difficult and costly, and greatly complicate verification, validation, and optimization of the codes.

In this report, we consider the process of modernizing reference phenomenology modeling codes using modern software methodologies and language features. The ultimate aim of this work is to propose a method for updating these codes, which improves the speed, reliability, maintainability, verification, and validation of the codes. We also address the creation of platform independent codes that can run on any computer system with the proper language compiler.

INTRODUCTION

The reference phenomenology modeling codes in use today are physics based codes. Most of these codes have multiple independent avenues of validation. While these codes are reasonably computationally efficient, they have structural and architectural shortcomings which include: poor and inconsistent documentation; "spaghetti string" code as a result of multiple corrections and "patches" to the code; randomly distributed hardwired constants; improper and inconsistent use of global variables; uneven scoping of subroutines and functions; lack of discernible architecture for user input handling and output file organization. The maintenance and validation problems caused by these shortcomings have prompted discussion of conversion of the entire corpus of science and engineering codes to more modern software methodologies and programming languages. This issue is becoming more urgent as languages and compilers evolve, since obsolete language constructions in older codes will be phased out over time.

It is considerably easier to talk about the benefits and advantages of such conversions than to actually perform them. In 1987, the astronomical community began one of these 'drives for modernization' and undertook a conversion of AIPS (Astrophysical Image Processing System ~400,000 lines of code) from a reasonably well designed and documented F77 original to C++. Nearly ***ten*** calendar years, and more than a ***hundred*** programmer years later, the 'new and improved' product "AIPS++" has still not been completed! [1]

In this report, we address multiple aspects of the code modernization issue, and discuss the advantages and disadvantages of the two most often suggested programming languages for phenomenological coding. In places where it is helpful to have a concrete example of a phenomenology code, we will use MODTRAN [2]. MODTRAN is a widely used code that solves the radiative transfer equations for propagation of radiation through the atmosphere. MODTRAN is reasonably large (40,000 lines) and suffers from many of the shortcomings mentioned above. Finally, we recommend procedures for code modernization based on our studies and work found in the literature.

THE NEED FOR MODERNIZATION

The reference phenomenological codes are needed in many studies. These codes are continuously being updated to meet the needs of new applications. As standards and systems change, it is becoming more and more costly to adapt the reference phenomenological codes to new applications. In addition, the need to verify and validate the codes after each change adds additional expense and time. Thus, the need to modernize the codes to more maintainable structures that are easier to validate is clear.

In the quarter-century since the earliest modeling codes were developed, hardware and software have evolved tremendously. In addition a number of computer languages have emerged or evolved.[3-6] Software design techniques and more rigorous documentation standards have made maintenance and enhancement easier and more routine. CASE (Computer Assisted Software Engineering) tools have been developed to eliminate some of the drudgery of developing large software systems. Improvements in languages include new, faster, better flow control features (e.g., new loop options and "case-switch" and "if-then-elseif" statements) and I/O libraries that remove the burden of excessive specification from applications programmers. Hardware has improved even faster than software: a typical modern system which can be found on any scientist's desktop today has roughly 100 times the RAM, disk storage, and CPU speed of a typical *mainframe* of the late 1970s. It is important to stress just how different the computational environment of the late 1990s is from that of the 1970s because many of the assumptions behind the interaction between the original codes and their runtime environments are simply not valid in the modern computational world. Coding to these assumptions causes noticeable degradation of performance as well as unnecessarily convoluted code. Systems developed for standalone mainframes in the 1970s with limited memory and storage space don't generally perform well in modern distributed computing environments because they weren't designed to take advantage of modern hardware or software.

This issue is common in software written in the 1960s and 1970s; the reference phenomenological codes are much like legacy codes in this regard. However, *these* codes continue to be revised and must be used even though the cumulative effect of years of continual and confusing code modifications on their reliability and performance cannot be

accurately assessed. The bulk of the modeling codes are still written in older dialects of FORTRAN (FORTRAN 66, FORTRAN 77, and FORTRAN 77 + Mil-Std-83 extensions) and use obsolete constructions (e.g., 'Hollerith constants' in string I/O). While the codes are generally efficient *computationally*, they don't follow modern software engineering practices. Many of the problems can be identified by a thorough review of the code and are relatively easy to fix. For example, implicitly typed constants can be explicitly declared, variables' scope can be changed from global to local, and i/o filenames can be made be changed to avoid name-clashes in distributed environments.

Collectively, fixing these problems would be time-consuming and tedious, but straightforward.

For codes that have not been altered since their original release, many of the problems could be solved by simply re-engineering the existing codes in their original language using modern software standards. Unfortunately, most of the codes have been extensively modified over the last two decades, and control flow and logic have become so obscure, and documentation so fragmentary, that the codes can no longer be clearly understood or easily tested. Thus, a complete redesign seems to be indicated.

A modular redesign of the modeling codes would involve a complete restructuring of the code to encapsulate I/O and data operations, documentation, and unit and system test code. Increased system resources and new design and coding techniques and constructions mean that, where appropriate and practical, GUIs can be used, all files need not be written out to disk, and code should operate automatically in distributed network environments. It may also be possible to standardize the user interfaces so that expertise with one model can be more easily transferred to another, and standardize and document input and output formats to make it easier to compare results from different modeling codes. Intelligent use of CASE tools would help a reengineering team decide where to concentrate code optimization efforts, so that the most computationally intensive routines would be as efficient as possible. As part of modernization effort, new software practices encouraging design reuse and code reuse should be used. Design decisions, control flow, functionality, test suites, bounds testing, and exception handling/trapping should be documented before and during code development. Existing code that is reused must be quality-tested and documented to current standards. Design goals would include maximizing the reusability,

readability, maintainability, and portability of the new modeling codes, accurate and complete documentation, and development of validation test suites.

Redesign of reference phenomenology codes must focus on several goals:

1) the codes must be accurate and reliable

2) they must be computationally efficient

3) they must operate across platforms

4) they must be easy to modify and maintain

5) where possible they should make use of multithreading and parallelization.

TECHNIQUES OF MODERN PROGRAMMING

In order to discuss the techniques of modern programming, we must first consider the languages that one is to program in and realize that there are a multitude of platforms that the codes may eventually operate on. With this in mind, the need to work with standards becomes evident. Since most platforms have compilers for the standards of a language, codes written in the standard can operate anywhere. More often than not, the use of extensions of a language in coding leads to problems when the codes are ported to another platform.

The reference phenomenological modeling codes are almost exclusively written in old styles of FORTRAN, even though programming languages have evolved and many useful new features have appeared in the last twenty years. These new features, found in upgrades of old languages (e.g., FORTRAN90[7, 8], C++[9]) as well as entirely new languages (e.g., Eiffel[10]), make it much easier to write self-documenting, clean, and more easily maintainable codes.

The degree to which structured programming can be achieved is dependent on the programming language. While there is a significant amount of research in progress to

invent the best language for structured programming, the languages that are thought to be most suited for structures programming are those which contain the basic building blocks for creating structured programs.[11] Among these languages are PL/I, PL/C, PASCAL, ALGOL, C/C++, and modern dialects of FORTRAN.

There are several major areas where substantial improvements have occurred that we will mention here {see Appendix A for further information}.

Control Structures

Early programming languages generally had two basic ways to repeatedly execute a block of code: *counted loops* (DO I=1,10) and *conditional loops* (DO WHILE), and a simple IF/THEN/ELSE construction. GOTO and *computed* GOTO statements were available. (Computed GOTOs allowed a programmer to jump to one of several labeled lines depending on the value of a specified variable.) These limited control structures meant that certain operations required difficult programming structures.

Modern programming languages have expanded the IF/THEN/ELSE statement to IF/THEN/ELSEIF/ . . . /ENDIF, thus making several consecutive tests possible in one block of code. In addition, there is a SWITCH or SELECT construction in all modern programming languages, and some languages (IDL or F90 / F95[12] / HPF[13]) provide a WHERE/ELSEWHERE/ENDWHERE control structure for matrix or parallel operations.

User-Defined Types

Early programming languages generally allowed for data of types REAL, INTEGER, CHARACTER, and STRING (group of characters). Thus, to store related information of different types a programmer usually had to create two arrays and be sure that the data were stored and read with identical offsets for the different data. Modern languages allow the creation of 'user-defined' data types (Records or Structures), which can hold multiple instances of several types of data in one logical unit, making it much easier to manipulate (add data to / extract data from / pass to and from subroutines, etc.) Many languages allow for Unions, or Variant Records, so that the information within a structure or record need not always be in the same form.

Operator and Function Overloading

As programming languages have been developed and refined, the number of operations that can be performed on various types of data has been expanded and standard operators have been expanded or "overloaded". Modern languages not only have more built-in operators, but also allow for the expansion of standard operators to user-defined types. For example, in C/C++ there is no built-in complex number type, so programmers must define the type "complex" as containing two real numbers, often called "Re" and "Im", and then define how the arithmetic operations '+', '-', '*', and '/' are defined for complex numbers. Once that is done, one may write 'c = a + b' and get meaningful results whether a, b, and c are complex numbers, strings, integers, reals, etc.

In addition, the concept of "function overloading" is well developed in modern languages. It is not hard to imagine a need for a function "plot" that could be defined for all standard types of numbers and also the user-defined types "complex," "3Dvector", etc. One could call this "plot" function and get meaningful outputs for different kinds of input without calling different type-specific functions or passing a "type" argument. This feature can be used to simplify program structures by moving the details of complex calculations to lower-level functions or libraries.

Dynamic Memory Allocation

Early languages forced programmers to decide in advance how much memory to allocate to data sets and a number of variously effective techniques were used to assure that data didn't exceed its allocated storage. Modern languages allow one to dynamically allocate and de-allocate memory storage on demand. This is a very powerful feature that can cause serious and subtle bugs if programmers are not careful to allocate correct blocks of memory and de-allocate them when they're no longer needed, and "program defensively" to be sure that allocation and de-allocation work as expected in different operating environments.

Object-Oriented Analysis, Design, and Programming

Object-orientation is a paradigm that evolved in the late 1970s and throughout the 1980s to facilitate the design and construction of large, complex, trustworthy software systems. Object-orientation requires a shift from designing and programming algorithms to

{construct} a set of interacting autonomous "objects" at the level of system analysis, design, and programming. The strength of object-oriented methodologies is that they {coerce} the minimization of interactions between separate parts of a system and reduce the possibility of unintended consequences. The major drawbacks of these methodologies are that it will be difficult and time-consuming to re-engineer old codes using object-orientation, and the result will probably be a slower code especially for computation-intense sections of code. {See Appendix B for more information.}

Distributed Multi-Threaded Computing

As networked groups of workstations and personal computers have replaced centralized mainframes, new techniques for controlling the interactions between autonomous computers and between applications and operating systems have developed. In particular, the possibility of using Threading[14-16] and Distributed Computing[17] should be kept open during the re-engineering of phenomenological modeling codes.

The basic idea behind threading is that even on a single-CPU system there will be times during program execution when the CPU is sitting idle waiting for various I/O processes to finish. It is possible to organize code to keep the CPU working during those times and thereby improve execution speed by organizing the program into multiple 'threads' of activity that can act cooperatively. Multithreading is an important and powerful technique with multi-processor systems. A desktop system with 2 or 4 CPUs can typically achieve nearly 2x and 4x improvements in performance for properly designed and implemented code.

Distributed computing techniques have evolved, as networked small computers (PCs and workstations) have become more common. Virtually any workplace now has its own local area network (LAN) and most sites have connections to the global Internet now. Over the past decade several standard techniques for distributing processing load across networks of computers have developed. These include Sun Micro-System's Remote Procedure Call (RPC) standard {references} and OSF's Distributed Computing Environment standard (OSF/DCE) {references} at the system software level and the so-called message passing interfaces such as MPI {references} and Oak Ridge National Laboratory's PVM[17].

See Appendix C for more information.

LANGUAGE ISSUES

The programming language used has a considerable effect on software reliability: in general, the "higher" the level of language, the fewer the errors. While machine language programming can be the fastest computing method, it is platform specific, extremely time consuming to write, prone to errors, and extremely difficult to debug. High-level languages eliminate several levels of software errors by hiding the machine idiosyncrasies and allowing any given function to be expressed in fewer statements. Programs in high-level languages are more understandable, easier to change, allow the compatibility and portability of programs, and can be self-documenting. The biggest advantage of high-level languages is their ability to express and manipulate complex data structures. These points have been confirmed in a combined Bell Laboratories, General Electric, and MIT study.[18]

We need to ask, "What is the best high-level programming language for the modernization of the reference phenomenology codes?" To answer this question, we will compare and contrast the two leading modeling languages, FORTRAN and C/C++, on several points relevant to the modernization issue.

At the time of this report, the reference phenomenology codes are still largely implemented in older FORTRAN (F66, F77). It is certainly possible to find people who are comfortable working with F66, but we expect that as time goes on, it will become increasingly difficult to find qualified people to maintain the codes, as more and more of the community shifts to newer, more modern programming languages.

The long-term solution to this problem is evidently to change the language in which the codes are written. *But which computer language should be used for code modernization?* While there is a plethora of languages from which one might choose, the truth is that choosing a language that scientists and engineers don't often use, or which will be painful for them to learn, simply isn't realistic. FORTRAN and C utterly dominate the Scientific – Engineering – Numerical programming landscape and will do so for many years to come.

The focus of this section then will be to consider the question of whether it is better to modernize the phenomenology codes by upgrading the codes to modern FORTRAN (F90, F95, HPF, F2000 etc) or by undertaking a wholesale translation to C (or possibly C++).

Such a translation would take a great deal more time and effort than the FORTRAN-only path, but if there are compelling reasons in favor of C/C++, then that may be the more prudent approach.

Below we investigate some of the relative merits and shortcomings of C/C++ and FORTRAN with regards to the modernization issue and address several points often raised in discussions of reference-code modernization that seem relevant in reaching a decision about the proper modernization path:

- Historically, most numerical work has been done in FORTRAN. CS people don't usually have much experience with FORTRAN, and generally do their own work in C or C++. Why does this dichotomy exist, and is it relevant to the modernization discussion?

- Which language will generate more efficient executable code? Is there a speed reason for converting code from older FORTRAN into C/C++ instead of modern FORTRAN?

- In which language will it be easier to maintain the modernized code in? Is there a code maintenance reason to prefer C/C++ to modern FORTRAN or vice-versa?

- In discussions of code modernization one also often hears the virtues of object oriented design extolled. Should the phenomenology codes be re-engineered so as to be object oriented? And, if so, does that mean the codes should be rewritten in C++?

- Modern computing uses distributed processing and multithreading to bring the power of multiple processors and workstations to bear on problems. People generally seem to access these features from C. Is it *only* possible to use these features from C/C++? And, if so, is this a reason to prefer using C/C++ for code modernization?

- Another feature of modern computing is to write code that does computation in parallel to make use of multiple processors. Such parallel processing [24] code is particularly effective in reducing the computation time in many number-crunching applications. Is this feature of parallel processing accessible from FORTRAN or C/C++?

The balance of this section on languages will address the answers to these questions.

The Dichotomy in Communities

Scientific programmers tend to be concerned with creating algorithms that accurately reflect physical phenomena. The programs they create are most often written in FORTRAN and are often intended to be used (by their authors) to solve specific problems and not by a community of general users. Clean I/O, documentation, maintainability, and (what in the CS community would be considered) good programming style have historically been sacrificed in favor of computational efficiency, accuracy, and speed of development.

For the most part, Computer Science (CS) departments don't tend to be involved in Numerical or Scientific Computing. Instead, they tend to be involved in Systems Programming, Graphical User Interfaces (GUIs), Object Oriented Programming/Design (OOP/D), and Management Information Services (MIS). These interests lead to a concentration of effort in languages well suited for systems applications and text manipulation (e.g. C/C++ and database languages) with much less attention being paid to numerical languages. Most CS departments don't teach programming (much less *numerical computing*); instead, it is common to teach *survey* courses, which spend a few class sessions on the *features* of a dozen or more programming languages. If pinned down, CS people will assert that FORTRAN is *old-fashioned* or *archaic* and will often cite a litany of features[19-21] that F66 and F77 lack. *Older* dialects of FORTRAN did lack many programming amenities, but one must remember that FORTRAN's *raison-d'être* has always been floating point computations, rather than systems programming, and many language features were sacrificed in the name of efficiency. The situation is very different for the modern dialects of the language.

Modern FORTRAN is a term used within this report to refer to the family of programming languages that has grown out of the older dialects of FORTRAN in which the phenomenology codes were written. Full ANSI-ISO standards for FORTRAN 90 (F90) and FORTRAN 95 (F95) exist, and standards for High Performance FORTRAN (HPF) and FORTRAN 2000 (F2000) are being drafted. This standardization of language is a key difference with C/C++. There is an ANSI standard for C but not for the C runtime library, and the language standard is typically NOT fully implemented; there are no ANSI standards

for C++ or any of the other more specialized C dialects (e.g. data-parallel C, concurrent C++, etc).

In the table below are listed the features and whether they are supported by C/C++ or by modern FORTRAN dialects[7, 8,22-24]

The latter features in the table greatly facilitate coding in modern multiprocessor/parallel computing environments, and are not available in *any* other (ANSI or ISO) standard general purpose programming language: they are *unique* capabilities of modern FORTRAN.

To summarize: CS people prefer C to FORTRAN because the domain in which their applications mostly lie (Systems Programming, GUIs, Databases, Compiler Building) are ones which are dominated by integer/character performance rather than numerical performance and flexibility. When people speak of FORTRAN's shortcomings, they are almost always speaking about archaic dialects of the language. We conclude that:

THE CS COMMUNITY'S ATTITUDE TOWARDS FORTRAN IS LARGELY BASED ON UNFAMILIARITY AND IS THUS NOT A VALID REASON TO CONVERT NUMERICAL REFERENCE CODES FROM FORTRAN TO C/C++

Table 1 Features of modern programming languages and existence in C/C++ or FORTRAN.

Feature	C/C++	modern FORTRAN
dynamic memory allocation and de-allocation	supported	supported
pointers	supported	supported
encapsulation and data hiding	supported	supported
structures and user defined data types	supported	supported
operator and function overloading	supported	supported
recursion	supported	supported
switch/case constructs	supported	supported
direct manipulation of arrays and matrices and associated sub-units as objects	not- supported	supported
Intrinsic Auto-parallelization	not- supported	supported(HPF only)
intrinsic and user-defined "Elemental" functions	not- supported	supported(F95, HPF, F2000)
intrinsic functions for manipulation of arrays (dot_product, shape, size)	not-supported	supported
intrinsic functions and operators for manipulation of complex numbers	not-supported	supported

The Execution Speed Issue

It is often supposed by people outside of the science/engineering community that languages like C/C++ are somehow inherently more efficient than languages such as FORTRAN, and that converting older codes from FORTRAN to C/C++ will result in dramatic speed increases. There is some truth to this in applications that primarily manipulate text strings and short integers, but it is NOT true for floating point intensive programs like the phenomenological reference codes.

In fact, *for numerical applications*, FORTRAN should theoretically always be at least as fast as (and usually faster than) C/C++ for the following reasons.

1) Most implementations of C/C++ don't allow programmers to use single precision; real numbers are cast to double precision internally and then cast back to single for I/O! (The ANSI C standard theoretically allows for this to be overridden, but the capability is rarely implemented.). This means that on certain hardware platforms (see discussion below) programs that are dominated by single precision floating point arithmetic will run 30% to 50% faster under FORTRAN.

2) C is harder to optimize[25] because of its array syntax. C implements arrays using pointers. This is a more efficient approach for memory usage, but less efficient for computational speed. Pointer references cannot readily be relocated in loop and control structure optimization because of the danger that pointer arithmetic might be corrupted.

3) C/C++ lacks a series of built-in intrinsic functions for numerical work; C *does* have transcendental functions and so forth implemented as standard *library* functions, but they cannot be readily in-lined and optimized in most implementations of the language.

The reader may have noticed that these reasons have to do with compiler optimization and underlying hardware platform. Any platform that supports single and double precision floating point AND has a globally optimizing FORTRAN compiler should show a clear speed advantage for FORTRAN. This is further demonstrated below.

- Most mainframes, and scientific workstations (Sun's, SGI's, HP's, DEC Alpha's, etc) meet these requirements and *do* show FORTRAN as being substantially faster.

- The Intel architecture,[26] which is common to PC's and clones, does NOT meet these requirements and does NOT show a substantial FORTRAN speed advantage. This is because:
    - The Intel hardware performs floating point in 80 bit IEEE extended double precision and then casts results to 32 bit IEEE single or 64 bit IEEE double – this means that there is very little 'hardware speedup' in using Single precision, hence FORTRAN single precision isn't a big advantage.
    - The Intel platform FORTRAN compilers[27,28] for the PC don't do optimization of the caliber one expects on a mainframe or workstation, hence there is little advantage taken of the global structure, loop, and in-lining optimizations mentioned above.

The upshot of these two points is that *on PC's and PC clones*, FORTRAN will be equivalent to C/C++. We have illustrated this by listing some benchmark results for a suite of C and FORTRAN compilers including Microsoft Visual C++[29], Watcom C++ [22], Microsoft FORTRAN Powerstation[29], Watcom FORTRAN 77[22], Digital Visual FORTRAN.[30]

The tests in the table were derived from Internet sources and various texts,[31-35] and the source code for each is listed in the appendices. The tests show manifest equivalence between C and FORTRAN, with one exception: the 'floats' benchmark. The 'floats' test consists of a tight loop that performs intensive floating point manipulations and heavy array indexing on a single precision array that is several thousand entries long. This is PRECISELY the kind of loop that C *cannot* optimize properly because of the way arrays are implemented (point 2 above). The C/C++ compilers that were tested were unable to devise efficient machine code for the routine, with the result that *on a PC* the routine ran 10 times faster under FORTRAN than it does under C/C++. We expect the difference here to be even greater on Workstations and high-end machines.

To summarize: There is a difference in execution speed dependent on the choice of vendor for a compiler, but this is generally small and can be ignored in the following discussion. From Table 2, we see that for many of the programs, FORTRAN and C are roughly the same speed on the PC. However, even on this platform, which favors C, the FORTRAN times are slightly faster with the exception of the TRIGS test. Once floating-point

operations are considered however, FORTRAN is a clear winner. The final factor is about a doubling of speed using FORTRAN. It is important to realize at this point that most of the reference phenomenology codes make heavy use of floating point operations.

Table 3 gives the results for a VAX 6610 computer system running OPENVMS 6.3. This is a true single/double precision machine and the results indicate this fact. While FORTRAN is 2.25 times faster than C on average on this computer, all of the FORTRAN times are significantly faster than the corresponding C times. Since most of the reference phenomenology codes use single precision arithmetic, a substantial speed increase could be realized for FORTRAN coding on some platforms. Table 4 gives a partial set of results for a 180 MHz r5k*1 processor running IRIX 6.3. At the time of the tests, the system did not have a FORTRAN 90 compiler installed. The two tests shown were for codes written in FORTRAN 77, which we could compile and test. Again the FORTRAN times are faster by factors of 1.65 and 2.40 for FLOATS and TRIGS, respectively. From the above results, we conclude:

FORTRAN NUMERICAL PROGRAMS EXECUTE FASTER, SOMETIMES MUCH FASTER, THAN C NUMERICAL PROGRAMS, THUS THERE IS NO SPEED ADVANTAGE TO BE GAINED IN CONVERTING NUMERICAL REFERENCE CODES FROM FORTRAN TO C/C++

TABLE 2. PC BENCHMARK RESULTS

BENCHMARKS	MS C/C++	WATCOM C++	MS FPS	WATCOM F77	DEC VF	BEST C	BEST FORTRAN	RATIO†
XAIRY	49.1 S	48.1 S	41.1 S	42.1 S	40.0 S	48.1 S	40.0 S	1.20
XAMOEBA	23.0 S	22.0 S	24.1 S	24.0 S	21.0 S	23.1 S	21.0 S	1.10
XBICO	32.1 S	33.0 S	28.1 S	27.0 S	27.0 S	32.1 S	27.0 S	1.19
XCONVLV	25.0 S	25.1 S	23.0 S	23.0 S	24.0 S	25.0 S	23.0 S	1.09
XDAWSON	18.1 S	18.0 S	15.0 S	16.0 S	16.0 S	18.0 S	15.0 S	1.20
XMEMCOF	27.0 S	27.0 S	28.0 S	26.1 S	24.0 S	27.0 S	24.0 S	1.13
XERFCC	32.0 S	31.1 S	28.0 S	27.1 S	30.0 S	31.1 S	27.1 S	1.15
XGASDEV	36.0 S	35.0 S	32.0 S	32.0 S	33.1 S	35.0 S	32.0 S	1.09
XROFUNC	22.0 S	22.0 S	21.1 S	21.1 S	21.1 S	22.0 S	21.1 S	1.04
TRIGS	31.7 S	29.7 S	30.8 S	30.8 S	30.8 S	29.7 S	30.8 S	0.96
FLOATS	256.5 S	261.0 S	26.0 S	22.0 S	26.0 S	256.5S	22.0 S	11.66
RADTRAN	31.5 S	33.5 S	25.6 S	24.8 S	23.4 S	31.5 S	16.8 S	1.88
Total Time	584.0 S	585.5 S	322.0 S	316.0 S	316.4S	579.1S	306.4S	1.89
RS*	.53	.53	.95	.97	.97	.53	1.0	

* Relative Speed is defined as the total time ratio of the fastest calculation to the current calculation. Thus, the smaller the number the slower the code ran with the given compiler.

† Ratio is C time divided by FORTRAN time; hence, a value greater than 1 indicates FORTRAN is faster in the numerical computation.

Table 3. Results for a VAX6610 running OPENVMS 6.3

FUNCTION	VAX-C	VAX-F77	RATIO†
XAIRY	25.8 s	16.6 s	1.55
XAMOEBA	16.2 s	9.8 s	1.65
XBICO	24.5 s	18.1 s	1.35
XCONVLV	17.4 s	10.1 s	1.72
XDAWSON	18.2 s	12.5 s	1.46
XERFCC	26.5 s	18.6 s	1.42
XGASDEV	62.0 s	34.9 s	1.78
XMEMCOF	43.6 s	25.9 s	1.68
XROFUNC	20.6 s	12.1 s	1.70
FLOATS	398.0 s	116.2 s	3.43
TRIGS	51.2 s	37.7 s	1.36
TOTAL	704.0 s	312.5	2.25

The numerical recipes times are for 400 iterations of the primitives, while the 'floats' and 'trigs' benchmarks are for a single iteration

† Ratio is C time divided by FORTRAN time; hence, a value greater than 1 indicates FORTRAN is faster in the numerical computation.

Table 4. Results for a 180 MHz r5k*1 processor running IRIX 6.3

FUNCTION	C	FORTRAN
XAIRY	10.1 s	NA
XAMOEBA	9.6 s	NA
XBICO	10.0 s	NA
XCONVLV	8.9 s	NA
XDAWSON	9.1 s	NA
XERFCC	16.2 s	NA
XGASDEV	10.1 s	NA
XMEMCOF	17.8 s	NA
XROFUNC	9.2 s	NA
FLOATS	9.9 s	6.0 s
TRIGS	73.9 s	30.8 s

The numerical recipes times are for 400 iterations of the primitives, while the 'floats' and 'trigs' benchmarks are for a single iteration

The Code Maintenance Issue

We accept *a priori* the need for a well-defined programming style to which code developers should adhere!

Once a consistent programming style and discipline has been adopted for all code development and rigorously enforced, then any sufficiently modern programming language is equally suitable from a software engineers' standpoint. Of course the term "sufficiently modern" needs to be clarified. We take this to mean any language possessing the control and data structures and flexibility to support the types of applications programming indicated; certainly any modern programming language meets this criterion, including the dialects of modern FORTRAN or C. This issue aside: are there other maintenance reasons to prefer one language to another?

The community that originally developed the phenomenological models is unquestionably a FORTRAN community. Since FORTRAN was expressly created to facilitate the translation of scientific/engineering formulae into computer code, it seems clear that it is well suited for that purpose. C on the other hand was not intended to even deal with floating point numbers (they were added after the fact[36]) and lacks operators for fundamental operations such as exponentiation. Unless one can show some *compelling* maintenance reason to prefer code written in C/C++ over FORTRAN, one should be extremely cautious about undertaking such a conversion.

In fact, there are compelling reasons for NOT making a translation. If new code development work is done in modern FORTRAN the very same compilers can be used for both old F66/F77 code and new code written in F90/F95/HPF (F66/F77 is still supported by the new compilers). This vastly simplifies verification and validation. Individuals sometimes raise the subject of FORTRAN to C translation programs here attempting to claim that C compilers can play this same role for the old FORTRAN code. There are FORTRAN to C translation programs which are reasonably robust (e.g. FSF f2c[37]) but they generally produce intermediate C code that is nearly unreadable and that is substantially slower than native C or FORTRAN code would have been. These translation programs

typically have problems with complex codes. F2c does NOT successfully produce a binary for MODTRAN for example, even though no warning or error messages are generated.[38]

Furthermore, there are ***no*** useful FORTRAN to C++ translators since off the shelf AI technology isn't up to parsing code and applying human-expert-level knowledge of the application domain to generate appropriate and efficient classes and objects.

It should also be pointed out that the technique of taking a program and converting it to an equivalent program in another language or in an improved version of the same language, for example see the *Ashcroft-Manna technique*,[39] often has a negative effect on reliability. Methods such as *step-wise refinement* of Dijkstra[40] or later enhancements of the method[41,42] have been shown to be a better approach to modernizing programs.

To summarize: It is expected that whichever language is chosen for modernization, there will be a set of well defined coding guidelines that are adopted and adhered to. Given such guidelines, the question of which language affords the easier maintenance then boils down to a question of familiarity and validation/verification. The phenomenology codes were written in FORTRAN by a FORTRAN community. Modernized FORTRAN code can readily be checked against legacy code with the very same compilers. C compilers cannot provide such a facility, and the C programmers used to modernize the codes will almost certainly not be scientists who understand the essence of the codes. We conclude:

KEEPING NUMERICAL REFERENCE CODES IN A FORTRAN DIALECT WILL MAKE IT EASIER TO MAINTAIN, ENHANCE, AND VALIDATE THEM IN THE FUTURE

The Object Orientation Issue

Both modern C/C++ and FORTRAN can be used to write object-oriented code. It is true that FORTRAN lacks a specific keyword for class, but FORTRAN has entities called modules that perform the same role. Also, note that things such as operator and function overloading, scoping, private and public functions, and data hiding are neither derived from Object Orientation nor are they dependent on Object Orientation. They are simply features of modern programming languages that OOP often makes use of. Most modern languages

support these features: and both C++ and modern FORTRAN are sufficiently robust for implementing an object-oriented design. Hence, we conclude:

BOTH MODERN FORTRAN AND C/C++ CAN BE USED TO WRITE OBJECT ORIENTED CODE, HENCE ACCESS TO OBJECT ORIENTED DESIGN IS NOT A VALID REASON TO CHOOSE ONE LANGUAGE OVER THE OTHER

Distributed and Multithreaded Computing

Distributed computing and multithreaded computing are two methods that allow codes to take advantage of multiple processors or systems. Utilizing these methods on an n processor system or across n systems can lead to reducing computation time by almost $1/n$. Are these features dependent on a particular language?

Threading is an operating system level function and can be readily accessed from any robust modern programming language. Making a thread_create() call is similar in either C/C++ or F95. Distributed computing is implemented in several ways on different systems. Generally speaking, there is an operating system interface layer which uses TCP/IP, and possibly other network protocols to exchange packets between different computers, and a applications programming layer which supports a set of library calls that user programs invoke to interact with the system. In principle, nothing prevents users from making these calls from any language they choose to program in.

That said, we note that the people who maintain threading subsystems and distributed computing environments are generally CS systems programmer types who work in C and are not terribly concerned with providing other-language support for interfaces and libraries. Compiler vendors however do routinely provide interface libraries and support for threading. (For example, all of the C and FORTRAN compilers tested above for the PC for example came with full support for multi-threaded programming.). Distributed computing support is spottier from both compiler and systems vendors, but there are readily available *standard* systems, such as PVM or MPI, which come with C and FORTRAN interfaces, and are in extremely wide use throughout the world in distributed computing environments.

To summarize: Multithreading is a technique that can be readily accessed from most modern programming languages. Distributed computing support is spottier, but certainly there is extensive support available for both C and FORTRAN. We conclude:

BOTH MODERN FORTRAN AND C/C++ CAN BE USED TO WRITE MULTITHREADED AND DISTRIBUTED CODE, HENCE ACCESS TO THESE FEATURES IS NOT A VALID REASON TO PREFER ONE LANGUAGE OVER THE OTHER

Parallel processing

Parallel processing is a technique that is used to allow the computational components of a code to execute in parallel on systems with multiprocessors. For certain number-crunching algorithms the execution speed increase is a factor approximately equal to the number of processors. Taking advantage of this feature meant at one time that code had to be specially written for parallel optimization. This becomes a problem when applications need to operate on different platforms. With the new generation of FORTRAN compilers this is no longer the case. Using the High Performance FORTRAN compiler, parallel processing commands begin with "!HPF$" . The HPF compiler recognizes this and produces code optimized for parallel processing. FORTRAN compilers on other platforms see the exclamation point and treat the line of code as a comment line. Thus, the same code can be utilized across platforms. Multiprocessor platforms that have the HPF compiler can make use of parallel processing and other platforms simply run the code on a single processor. Another feature of this approach is that there is no time penalty for platforms that do not use HPF. Platforms that do use the HPF compiler simply have increased efficiency.

Discussion

We have considered each of the questions raised above. One of the questions (Dichotomy) seems to be irrelevant to the issue at hand. Two of the questions (Object Orientation, and Distributed Multithreaded Computing) are well addressed by both C/C++ and modern FORTRAN. The remaining three questions (Execution Speed, Code Maintenance, and

Validation/Verification, and parallel computing) are better addressed by modern FORTRAN.

Since undertaking a translation to C/C++ will be very costly and arduous, and since there are no evident reasons to prefer such a translation, it appears obvious that a modern FORTRAN dialect (F90 or F95, possibly with HPF extensions) should be used in modernizing the codes. This choice will generate code that is faster on most platforms, more portable, more easily maintainable, easier to verify/validate, and more familiar to the user community.

SUMMARY AND RECOMMENDATIONS

If the numerical phenomenology codes are to be modernized, it then seems essential to adopt a set of prioritized goals to guide the modernization effort. In order of decreasing importance, the appropriate set of priorities would appear to be:

- Accuracy and Reliability – The codes must be accurate. They must give correct answers given correct inputs. They must not allow incorrect answers to be produced even given incorrect or inconsistent inputs. They must give the SAME answers on different platforms in a consistent fashion.

- Performance - The codes must produce their answers as efficiently as possible. This most often means (but is not restricted to) in the shortest possible time. It might also mean with the least possible usage of total computer resources – disk and memory as well as CPU usage.

- Portability - The codes must efficiently produce accurate and reliable answers on multiple platforms. At the very least the codes must run on super-computers, high-end workstations, and low-end desktop systems. As far as possible, the codes should NOT be tied to particular hardware architectures, compiler implementations, or operating system environments.

- Ease of Maintenance and Modification - The updated design of the codes should make it easier to maintain the codes and implement future updates and improvements.

- Distributed-, multithread- and parallel-computing - Where possible distributed computing, multithreading and parallel processing should be taken advantage of. This must proceed in a platform independent manner to be useful in the reference phenomenology codes.

In order to achieve these goals, we recommend the following steps be taken to implement a code modernization.

1. General Considerations

The updating of codes must remove all compiler differences such that the results obtained from the code are independent of the compiler used. The codes should be independent of the platform they are executed on, for example see the *Posix* standard.[43] Thus, programming "tricks" that make use of compiler or platform specific characteristics should be avoided. This is particularly true if one considers the rapid changes in computer hardware that are occurring routinely.

Codes should be modularized and we should create or define objects common to codes. For example, with the Phillips Laboratory codes, we could create a weather object that defines fog, rain, clouds, etc. common to all the codes. We might also think about a geometry object to define the ray path, an atmosphere object to define the state of the atmosphere, and so on. This approach would help to unify the codes and reduce the future maintenance and enhancing of the codes. This approach also would improve configuration and lifecycle management.

Standardization of the Input and Output of the codes must be done in a modular and generalized fashion. Some of the techniques of OOP/OOD could be used in this task. Perhaps an input object can be created for this task. For the PL codes, a generalized input module could be created that can operate with all the radiative transfer codes. By the use of an Expert System, the fidelity of the input could range from minimal to maximum with the ES checking for consistency and filling in any missing information. This would also allow for backward compatible input and output. The output should also be cast in a standard

format such as HDF or FITS to ease in distribution of the data. Custom translators can then put the data in a form particular to each user.

All new coding should be DIS and HLA compliant to assure accessibility of the reference legacy codes. This will lead to cleaner more versatile codes.

Finally, all changes should aim at reducing the compute time for each code. This should be done, however, while observing the other guidelines discussed here.

2. Adoption of a clearly defined validation/verification methodology

A validation protocol, which must include input from the code scientists, should be defined and published. Guidance from the literature should be exercised.[11] This would lead to the development of a set of *comprehensive* test suites. The suites must exercise the full space of run-time parameter options. Codes should be re-engineered to use expert systems (ES), to check user inputs and prevent unphysical calculations. A sequence of more exotic tests by domain experts should be tried, where there are deliberate attempts to force failure modes. Only after codes have passed these tests should they be released for general use! The distribution of the codes should also include a suite of validation input data for each code and the verified output to compare to.

3. Adoption of consistent styles for coding and documentation

All functions, routines, and support files should contain complete descriptions of their contents. This should entail *at the very least*: Pseudocode for all routines, full API definitions ("interface" blocks) for all routines. To facilitate portability, *makefiles*, should be used. Many of the suggestions made in Kernighan *Software Tools* [44] would be appropriate here, as well as the coding and style guidelines that the Free Software Foundation[37] (makers of the GNU compilers and tools) has implemented.

4. Adoption of a modern programming language and code re-engineering to make best use of that new language

As discussed in the previous section, we believe the legacy codes should be upgraded to a modern dialect of FORTRAN (F90/F95 with HPF extensions). The resulting code will be

more portable to other platforms than C/C++, faster on the majority of platforms, and will provide for *portable* and *transparent* parallelization for multiprocessor environments. In addition to this intrinsic parallelization, we suggest the use of multithreading and distributed computing in those circumstances that seem to warrant it. Care must be exercised to make multi-threaded code that is platform independent. The choice of a modern dialect of FORTRAN will also result in a great reduction in the cost of modernization of the code.

BIBLIOGRAPHY

1. AIPS team member, Private communication, 1997.

2. Ontar Corporation, 1995, *The MODTRAN Report*, Ontar Corporation, North Andover, MA

3. Garg, V.K., and Ramamoorthy, C.V., 1991, *Computer Languages*, **16,** 5 *"Concurrent C: a language for concurrent programming"*

4. Dewhurst, S.C., and Stark, K.T., 1989, *Programming in C++*, Prentice Hall, New York

5. Edelson, D., and Pohl, I., 1989, "C^{++}: Solving C's Shortcomings," *Computer Languages*, **14**, 137

6. Pohl, I., and Edelson, D., 1988, "A to Z: C Language Shortcomings," *Computer Languages*, **13**, 51

7. Redwine, C., 1995, *Upgrading To FORTRAN 90*, Springer-Verlag New York

8. Metcalf, M., and Reid, J., 1990, *FORTRAN 90 Explained*, Oxford University Press, Oxford

9. Stroustrup, B., 1991, *The C++ Programming Language*, 2nd Edition, Addison-Wesley

10. Meyer, Bertrand, 1992, *Eiffel, The Language*, Prentice Hall, New York

11. Myers, G.J., 1976, *Software Reliability, Principles and Practices*, Wiley Interscience, J. Wiley and Sons, New York.

12. Wagener, J.L., 1995, *ACM FORTRAN Forum*, **14**, 2, 1 "FORTRAN 95, X3J3/95-007r1 Committee Draft"

13. Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, G.L., and Zosel, M.E., 1994, *The High Performance FORTRAN Handbook*, MIT Press, Cambridge

14. Kleinman S., Shah, D., and Smaalders, B., 1996, *Programming with Threads*, Sunsoft Press / Prentice Hall

15. Lewis, B., and Berg, D.J., 1996, *Threads Primer: A Guide to Multithreaded Programming*, Sunsoft Press / Prentice Hall

16. Nichols, B., Buttlar, D., and Proulx-Farrell, J., 1996, *Pthreads Programming*, O'Reilly & Associates

17. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V., 1994, *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge

18. Corbato, F.J., "*PL/I as a Tool for System Programming*,"DATAMATION **15** (5), 68-76 (1969).

19. Riel, A.J., 1996, *Object-Oriented Design Heuristics*, Addison-Wesley

20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., 1991, *Object-Oriented Modeling and Design*, Prentice-Hall.

21. Buzzi-Ferraris, G., 1993, *Scientific C++: Building Numerical Libraries the Object-Oriented Way*, Addison-Wesley

22. Watcom International Corporation, 1995, *Watcom C++ and Watcom F77*, Watcom International, Waterloo, Ontario

23. Lutowski, R. 1995, *ACM FORTRAN Forum*, **14**, 4, 13 "Objected-Oriented Software Development with Traditional Languages"

24. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., 1996, *Numerical Recipes in FORTRAN 90: The Art of* ***Parallel*** *Scientific Computing*, 2nd Edition, Cambridge University Press

25. Bierman, K., 1996, *ACM FORTRAN Forum*, **15**, 1, 8

26 Intel, 1996, *Pentium Pro Family Developer's Manual*; Intel Corporation, Mt. Prospect, IL

27. Jones, R.K., and Crabtree, T., 1988, *FORTRAN Tools for VAX/VMS and MS-DOS*, John Wiley NY

28. Schmitt, M.L., 1995, *Pentium Processor Optimization Tools*, AP Professional, Boston

29. Microsoft Corporation, 1995, *Microsoft FORTRAN Powerstation Professional Edition & Microsoft Visual C++ Professional Edition*, Microsoft Corporation, Redmond Washington

30. Digital Corp, *DIGITAL Visual Fortran*, Digital Equipment Corporation, Maynard, MA.

31. Engeln-Muellges, G., and Uhlig, F., 1996, *Numerical Algorithms with FORTRAN*, Springer-Verlag, Berlin

32. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., 1986, *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing*, 2nd Edition, Cambridge University Press

33. Ehrlich, R., 1973, *Physics And Computers, Problems, Simulations, and Data Analysis*, Houghton Mifflin Company, Boston

34. Nakamura, S., 1993, *Applied Numerical Recipes in C*, Prentice Hall, Englewood Cliffs

35. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., 1990, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd Edition, Cambridge University Press

36. Kernighan, B.W., and Ritchie, D.M., 1988, *The C Programming Language*, 2nd Edition, Prentice-Hall, Ottawa, Canada.

37 Free Software Foundation, (*software available from prep.ai.mit.edu*), Boston, MA

38 Cobb, W.K., 1997, Modtran 3.1.2 was obtained from Phillips Laboratory and was compiled using FSF's "f2c" Fortran-to-C translator program under Linux 2.0.30. Modtran compiled without warning or complication. Several attempts to execute the resulting program were made, however it consistently crashed during execution. The problems seemed to be associated with a problem early in the computation complaining of inappropriate file-access modes. No such errors were observed on the PC Windows platform or on VAX/VMS.

39. Ashcroft, E. and Z. Manna, "*The Translation of 'GO TO' Programs to 'WHILE' Programs*," *Proceedings of the 1971 IFIP Congress*, Booklet TA-2. Amsterdam: North-Holland, 1971, pp. 147-152.

40. Dijkstra, E.W., "*A Constructive Approach to the Problem of Program Correctness*," BIT 8 (3), 174-186 (1968).

41. Wirth, N., "*Program Development by Step-Wise Refinement*," *Communications of the ACM* **14** (4), 221-227 (1971).

42. Wirth, N., "*On the Composition of Well-Structured Programs*," *Computing Surveys* **6** (4), 247-259 (1974).

43. Lewine, D., 1991, *POSIX Programmer's Guide, Writing Portable Unix Programs*, O'Reilly & Associates

44. Kernighan, B. W., 1976, *Software Tools*, Addison-Wesley,

45. Ambler, S.W., 1995, *The Object Primer: The Application Developer's Guide to Object-Orientation,* SIGS-New York

46 Booch, Grady, 1991, *Object-Oriented Design With Applications,* Benjamin/Cummings Publishing, Redwood City, CA

APPENDIX A

New Language Features:

New versions of FORTRAN contain features that simplify many common operations. For example, in MODTRAN, one common problem is the need to initialize large arrays (often to zero). This can be done much more easily with the new built-in feature allowing an entire array to be set to some value in one statement.

In DRIVER:{line135}

```
      COMMON ABSC(5,47),EXTC(5,47),ASYM(5,47)
        . . .
        DO 80 I=1,4
          DO 80, J=1,40
            ABSC(I,J)=0.
            EXTC(I,J)=0.
80          ASYM(I,J)=0.
```

This can be re-written as 3 assignment statements in FORTRAN90:

```
ABSC = 0.
EXTC = 0.
ASYM = 0.
```

This code will also eliminate a possible bug (it's hard to know the intent of the programmer here): the first code fragment does not re-initialize the arrays entirely. All 3 arrays are defined as size (5,47) but initialized only to (4,40), and there's no easy way to know whether that was done on purpose. It's possible the array was re-sized in the COMMON without changing the initialization loop counters.

Control Structures:

```
CWHEN MODTRAN RUN IS COMPLETE {DRIVER line 895}
      IF (IRPT.EQ.0) GO TO 900
      IF (IRPT.EQ.4) GO TO 400
      SALB = SALBS
      IF (IRPT.GT.4) GO TO 900
      GO TO (100,900,300,400), IRPT
900 STOP
      END
```

Assuming that the GO TO a section of code to read and process new input is re-written to be a read-input subroutine and a processing subroutine, this can be re-written in FORTRAN90 as:

```
CWHEN MODTRAN RUN IS COMPLETE
       DO WHILE (IRPT=1 .OR. IRPT=3 .OR. IRPT=4)
        SELECT CASE(IRPT)
          CASE (1)
            CALL READ_ALL
          CASE (3)
            CALL READ_GEOM
          CASE (4)
            CALL READ_FREQ
         END SELECT
         CALL DO_MOD_RUN
       END DO
       STOP
       END
```

User-Defined Types:

User-defined structures can make it easier to organize, access, and pass information: {DRIVER line 435}

```
      COMMON /CARD3/ H1,H2,ANGLE,RANGE,BETA,RO,LEN
Cvariables not declared: IPARM,IPH,IDAY,ISOURC,
CPARM1,PARM2,PARM3,PARM4,TIME,PSIPO,ANGLEM,G,
CISAVE1,ISAVE2,ISAVE3,ISAVE4,
CSAVE1,SAVE2,SAVE3,SAVE4,SAVE5,SAVE6,SAVE7
      . . .
      READ(IRD,1312) H1,H2,ANGLE,RANGE,RO,LEN
1312  FORMAT(6F10.3,I5)
      . . .
      READ(IRD,1320) IPARM,IPH,IDAY,ISOURC
1320  FORMAT(4I5)
      READ(IRD,1322)PARM1,PARM2,PARM3,PARM4,TIME,PSIPO,
&   ANGLEM,G
1322  FORMAT(8F10)
      . . .
      CALL COMPAR(IPARM,IPH,IDAY,ISOURC,PARM1,PARM2,
&   PARM3,PARM4,TIME,PSIPO,ANGELM,
&   ISAVE1,ISAVE2,ISAVE3,ISAVE4,SAVE1,SAVE2,SAVE3,          &
 SAVE4,SAVE5,SAVE6,SAVE7,LSAME)
      CALL SSOLA(IPARM,IPH,IDAY,ISOURC,PARM1,PARM2,
&   PARM3,PARM4,TIME,PSIPO,ANGELM,
&   ISAVE1,ISAVE2,ISAVE3,ISAVE4,SAVE1,SAVE2,SAVE3,          &
 SAVE4,SAVE5,SAVE6,SAVE7)
```

This could be written in FORTRAN90 (*without* implicitly typed variables) as:

```
MODULE MODTRAN_GEOM
  TYPE BASE_GEOM
    REAL :: H1,H2,ANGLE,RANGE,BETA,RO,LEN
    INTEGER :: IPARM,IPH,IDAY,ISOURC
```

```
      END TYPE BASE_GEOM
      TYPE PARAMS
        REAL::PARM1,PARM2,PARM3,PARM4,PSIPO,ANGLEM,G
      END TYPE PARAMS
      TYPE SAVES
        REAL::SAVE1,SAVE2,SAVE3,SAVE4,SAVE5,SAVE6,SAVE7
        INTEGER :: ISAVE1,ISAVE2,ISAVE3,ISAVE4
      END TYPE SAVES
    END MODULE MODTRAN_GEOM
    . . .
    USE MODULE MODTRAN_GEOM
    . . .
    TYPE BASE_GEOM :: GEO
    TYPE PARAMS :: PARA
    TYPE SAVES :: SVLIST
    READ ({IRD,*})GEO%H1,GEO%H2,GEO%ANGLE,GEO%RANGE,
   &GEO%RO,GEO%LEN
    READ({IRD,*})GEO%IPARM,GEO%IPH,GEO%IDAY,GEO%ISOURC
    READ({IRD,*})PARA%PARM1,PARA%PARM2,PARA%PARM3,
   &  PARA%PARM4,PARA%TIME,PARA%PSIPOPARA%ANGLEM,PARA%G
    . . .
    CALL COMPAR(GEO,PARA,SVLIST,LSAME)
    CALL SSOLA(GEO,PARA,SVLIST)
```

NOTE that COMPAR and SSOLA would need to be re-written to handle the data structure input.

Operator and Function Overloading:

{as above}

in DRIVER{line 418}

```
H1S = H1
H2S = H2
ANGLES = ANGLE
RANGS = RANGE
BETAS = BETA
et cetera
```

If the "=" operator is overloaded so that one data structure of type BASE_GEOM can be set equal to another, this code can be written:

```
CCREATE THE DATA STRUCTURE AND WRITE A ROUTINE TO ADD C TWO     SUCH
STRUCTURES TOGETHER
MODULE MODTRAN_GEOM
    TYPE BASE_GEOM
      REAL :: H1,H2,ANGLE,RANGE,BETA,RO,LEN
      INTEGER :: IPARM,IPH,IDAY,ISOURC
    END TYPE BASE_GEOM
    . . .
    INTERFACE OPERATOR (=)
      MODULE PROCEDURE GEOM_COPY
 END INTERFACE
    . . .
CONTAINS
C   OVERLOAD "=" FOR THIS ARRAY
    FUNCTION GEOM_COPY (ORIG_GEOM, COPY_GEOM)
      TYPE (BASE_GEOM) ORIG_GEOM,COPY_GEOM)
        COPY_GEOM%H1 = ORIG_GEOM%H1
        COPY_GEOM%H2 = ORIG_GEOM%H2
        COPY_GEOM%ANGLE = ORIG_GEOM%ANGLE
```

```
        COPY_GEOM%RANGE = ORIG_GEOM%RANGE
        COPY_GEOM%BETA = ORIG_GEOM%BETA
        COPY_GEOM%RO = ORIG_GEOM%RO
        COPY_GEOM%LEN = ORIG_GEOM%LEN
        COPY_GEOM%IPARM = ORIG_GEOM%IPARM
        COPY_GEOM%IPH = ORIG_GEOM%IPH
        COPY_GEOM%IDAY = ORIG_GEOM%IDAY
        COPY_GEOM%ISOURC = ORIG_GEOM%ISOURC
      END FUNCTION GEOM_COPY
    . . .
  END MODULE MODTRAN_GEOM
  . . .
  USE MODULE MODTRAN_GEOM
  . . .
  TYPE BASE_GEOM :: GEOM,SAV_GEOM
  . . .
  SAV_GEOM = GEOM
  . . .
CRESTORE OLD VALUES
  GEOM = SAV_GEOM
```

Dynamic Memory Allocation:

{as above}

NOTE that in MODTRAN documentation, ML is limited to maximum of 34, while in file PARAMETER.LIST, LAYER is defined as 61 - so using this to dimension arrays will make them nearly twice as big as can possibly be used!

in main routine DRIVER, read of CARD2A is:

```
C   CARD 2C USER SUPPLIED ATMOSPHERIC PROFILE
      READ (IRD,1250) ML,IRD1,IRD2,(HMODEL(I,7),I=1,5
1250  FORMAT 3I5,18A4
```

{is there a default value for ML?}

in subroutine FLXADD:

```
      INCLUDE 'PARAMETER.LIST'
in COMMON statements, define
      BTOP(LAYDIM)
      IB(11),IBND(11)
      TAUM(3,LAYDIM),TWGP(3,LAYDIM)
      DPJ(3,LAYDIM),DPJW(3,11)

      NG=ML
      NLAYRS=NG-1

      DO 40 N=1,NLAYRS
        BTOP(N)=PI*BBFN(TLE(N),V)
        SMDPJ=0
        DO 30 K=1,3
          TAUM(K,N)=0
          TWGP(K,N)=0
            DO 20 MOL=1,11
              IB=IBND(MOL)
              IF(IB.LT.0) GO TO 20
```

```
                W(IB)=DENSTY(IB,IKMX-N) &
                          *PL(IKMX-N)*GKWJ(K,MOL)
                TAUM(K,N)=TAUM(K,N)+W(IB)*CPIS(MOL)
                TWGP(K,N)+TWGP(K,N)+ &
                     W(IB)*CPIS(MOL)*DPWJ(K,MOL)
20          CONTINUE
            DPJ(K,N)=DPC
            IF(TAUM(K,N).NE.0)DPJ(K,N)=TWGP(K,N)/TAUM(K,N)
            SMDPJ=SMPDJ+DPJ(K,N)
30        CONTINUE
          DO 41 K=1,3
            DPJ(K,N)=DPJ(K,N)/SMDPJ
41        CONTINUE
40      CONTINUE
```

With dynamic array allocation, all these arrays(and a lot of others) could be made allocatable, and probably smaller:

```
        PARAMETER(NUM_BANDS = 11, REPS = 3)
        REAL, DIMENSION( : ),ALLOCATABLE::BTOP
        REAL DIMENSION (NUM_BANDS)::IB,IBND
        REAL, DIMENSION ( :,: ), ALLOCATABLE::TAUM,TWGP
        REAL, DIMENSION ( :,: ), ALLOCATABLE::DPJ
        REAL, DIMENSION(REPS,NBANDS)::DPJW

C     GET "ML" FROM CARD 2C


        NG=ML
        NLAYRS=NG-1

        ALLOCATE (BTOP (NLAYRS))
        ALLOCATE (TAUM (REPS,NLAYRS))
        ALLOCATE (TWGP (REPS,NLAYRS))
```

```
ALLOCATE (DPJ (REPS,NLAYRS))


DO 40 N=1,NLAYRS
   BTOP(N)=PI*BBFN(TLE(N),V)
   SMDPJ=0
   DO 30 K=1,3
      TAUM(K,N)=0
      TWGP(K,N)=0
et cetera
```

NOTE that programmers must remember to de-allocate arrays:

```
DEALLOCATE (BTOP)
DEALLOCATE (TAUM)
DEALLOCATE (TWGP)
DEALLOCATE (DPJ)
```

APPENDIX B

Basics of Object-Orientation

{put in whatever works of the OO stuff - need to choose a focus and level of detail}

Object-oriented software design[19, 45,46] is the process of creating a program built around *classes* of *objects* with *attributes*, *operations*, and *relationships* to other objects defined intrinsically. The object-oriented programming process begins with the definitions of classes rather than processes, as is done in traditional programming languages. Object-oriented class definition has 4 properties that should be understood: data abstraction and encapsulation, modularity, class hierarchy and inheritance, and polymorphism.

The data abstractions called classes are grouped in physical files, or *modules*; it is usually a system design goal to group the classes that are most closely related. This is done both to make it easier to understand the relationships between classes and objects, and to make maintenance easier, since grouping closely related modules means that it is more likely that a partial re-compilation would be sufficient when small changes are made.

To properly define a class one must define what it is made of/controls (its attributes), what it does (its operations), and how it appears to and relates to the outside world (its relationships). The process of creating the classes is called *abstraction*. Objects' attributes and the methods they use to perform operations are generally private, and can be accessed only through a public interface of relationships defined for the class. Although there are efficiency tradeoffs associated with this information hiding, it is generally an OOD goal to create classes such that details like how data are stored or updated are *encapsulated* in each class and all operations are performed via the interface defined for each class.

Classes and objects are defined with hierarchical relationships between them, and if a class is a *subclass* of another, it will *inherit* attributes, methods and relationships from its parent class. Further, some classes inherit attributes from more than one parent class: they have *multiple inheritance*. It is possible to override inherited characteristics by redefining them in subclasses.

An operation may be defined such that it can cause different results on different classes of object. This is known as *polymorphism* in object-oriented parlance. In a sense, this is similar to the concept of "overloading" an operator in various computer languages. (The operation "+" can be used with integers, real numbers, and strings in C/C++, and as discussed above, it can be overloaded to have meaning in relation to user-defined types.) In object-oriented languages, operations are made polymorphic by creating multiple *methods* of performing an operation. The definition of each class includes the method of executing the operator for the class. For example, a Move is defined in chess differently for each piece, so Move of a Knight uses a different method and has a different result than Move of a Bishop. In an object-oriented program, one could define Chesspiece to be a Pawn (which can Move a certain way), OR a Knight (which can Move a certain way), OR a Bishop (which can move a certain way), and so on. A Move of a particular Chesspiece (which would have been previously defined as a Pawn, a Knight, a Bishop, etc.) would cause the correct action without further specification.

Re-engineering a program into an object-oriented language is a complex undertaking.[20] It is likely that the phenomenological modeling codes will be particularly difficult to reengineer, because of their complexity and the current state of the codes, as discussed above. It would be tempting but disastrous to undertake the project piecemeal, starting with the worst code or the most frequently called routines. However, object oriented programming depends heavily on the completeness and correctness of the initial definitions of classes and class hierarchies (including attributes, interfaces, and operations) and these in turn depend on careful and complete definition of the problem to be solved. This programming paradigm, if used correctly, will produce software that is better designed and easier to test, more easily maintained, enhanced, and reused, and well-adapted to multithreaded and distributed processing. If the problem of slower execution could be solved, the use of object-oriented design and programming methods could be very useful for many projects.

Object Orientation of MODTRAN

Re-designing existing atmospheric modeling codes would involve an intense initial period of examination of both the existing codes and the users' procedures and expectations while executing them.[19, 23, 45] Before any modeling codes are re-designed, it will be important to

make some of the design decisions concerning the eventual suite / Expert System Supersystem. Inter-relationships, if any, between the modeling codes should be defined. In addition, the requirements of several/many of the modeling codes to be eventually included in this process should be evaluated, so that design decisions can be made using the widest possible information base. The initial effort will involve creating abstract base classes for the first target model MODTRAN, then refining them in collaboration with domain experts for other models so that they will be sufficient for all models. Each model will need to create concrete subclasses from the base classes, tailored to their particular areas of operation; there must be mechanisms for "translating" these to make them available to other models (e.g., a MODTRAN run which exceeds its frequency range should be able to pass the atmosphere and geometry information on to RADTRAN to complete the calculation). Some of the calculations may be coded as calls to existing calculation routines via "wrappers" that will isolate old non-OOP working routines from the new code. Base class definitions of attributes, methods, and relationships must be made carefully, with the design goals kept in mind. Data storage issues must also be addressed: There are many questions: can we design a storage algorithm flexible enough to be used by ALL modeling codes? how will access methods be defined? which will be base, and which derived, attributes? where will different data be stored?

As an example, for a redesign of MODTRAN, the initial classes would include, but not be limited to,

- an Atmosphere class might include attributes involving Temperature, Pressure, Boundaries, AerosolDescription, GasDescription, Scatter (single and multiple, Mie and Rayleigh scattering all have different calculation methods), and methods for replacing standard values with a user-defined atmosphere, calculating with and without various types of scatter being considered, use the Navy Maritime atmosphere model, and so on. The Atmosphere class could be defined as a superclass that contains the entire atmosphere description, or as a superclass that contains a description of one atmospheric layer (thus a complete description of "the atmosphere" would be a collection of layer descriptions).

- a Weather class might include attributes Climate, Season, Precipitation, Clouds, Fog, ExtremeEvents (like hurricanes, tornadoes, of thunderstorms), Windspeed?, OceanProximity?, etc., and methods to let UserDefineCloud, ChangeToArmyVSAModel, SetCloudType, SetSeason, and so on.

- a Geometry class would include attributes involving the locations of the sun and moon, the time of day and time of year, the LOS start and end points in three dimension, and so on, with methods for allowing sun and moon location and LOS specification using different parameters, SetTimeOfDay, SetDayOfYear, and so on.

- a Frequency class that would have attributes of start and end frequency, interval, slit function.

- a "calculation" object (or two) that would multiply inherit from these four and use the information to calculate transmission or radiance - this would get atmosphere definition from Atmosphere, information on weather-dependent alterations of the atmosphere from Weather, location of sun and moon, time info., and LOS info. from Geometry, and a frequency info from Frequency. This should be able to decide out-of-bounds for model and package up complete input to forward to another model.

Current modeling code designs assume a continuous sequence of operations. To take maximum advantage of multithreading and distributed computing, our design phase must include consideration of the occasions of concurrent calculations, e.g., include a user-defined set of minor gases in the atmosphere object and add cumulus clouds to the weather object. These two objects can be changed concurrently (although both changes must be complete before the information is used by the radiance/transmittance calculation), and should be completely independent by design.

APPENDIX C

Multithreading and Distributed Computing

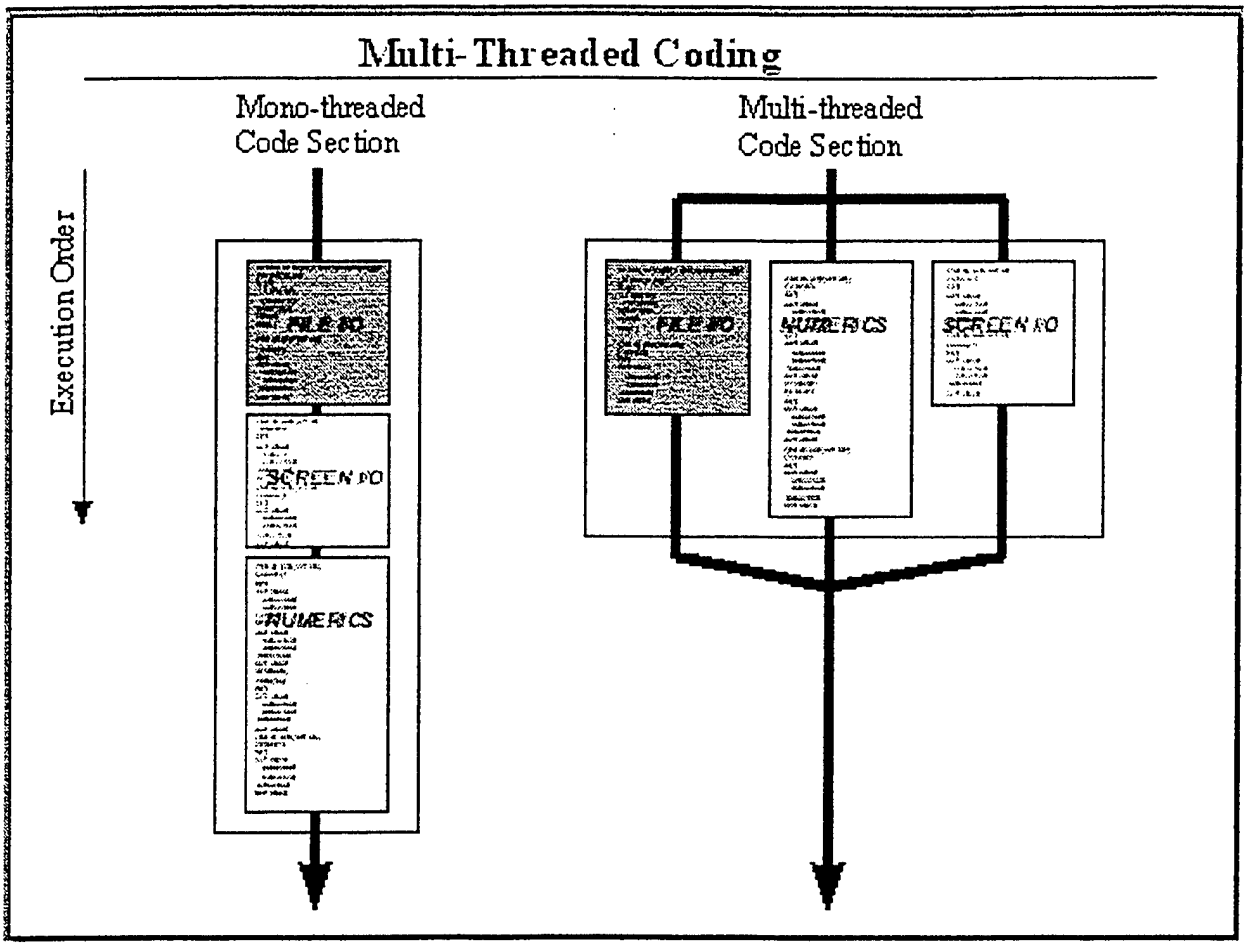


FIGURE C-1. A SCHEMATIC REPRESENTATION OF THE IDEA BEHIND MULTI-THREADING.

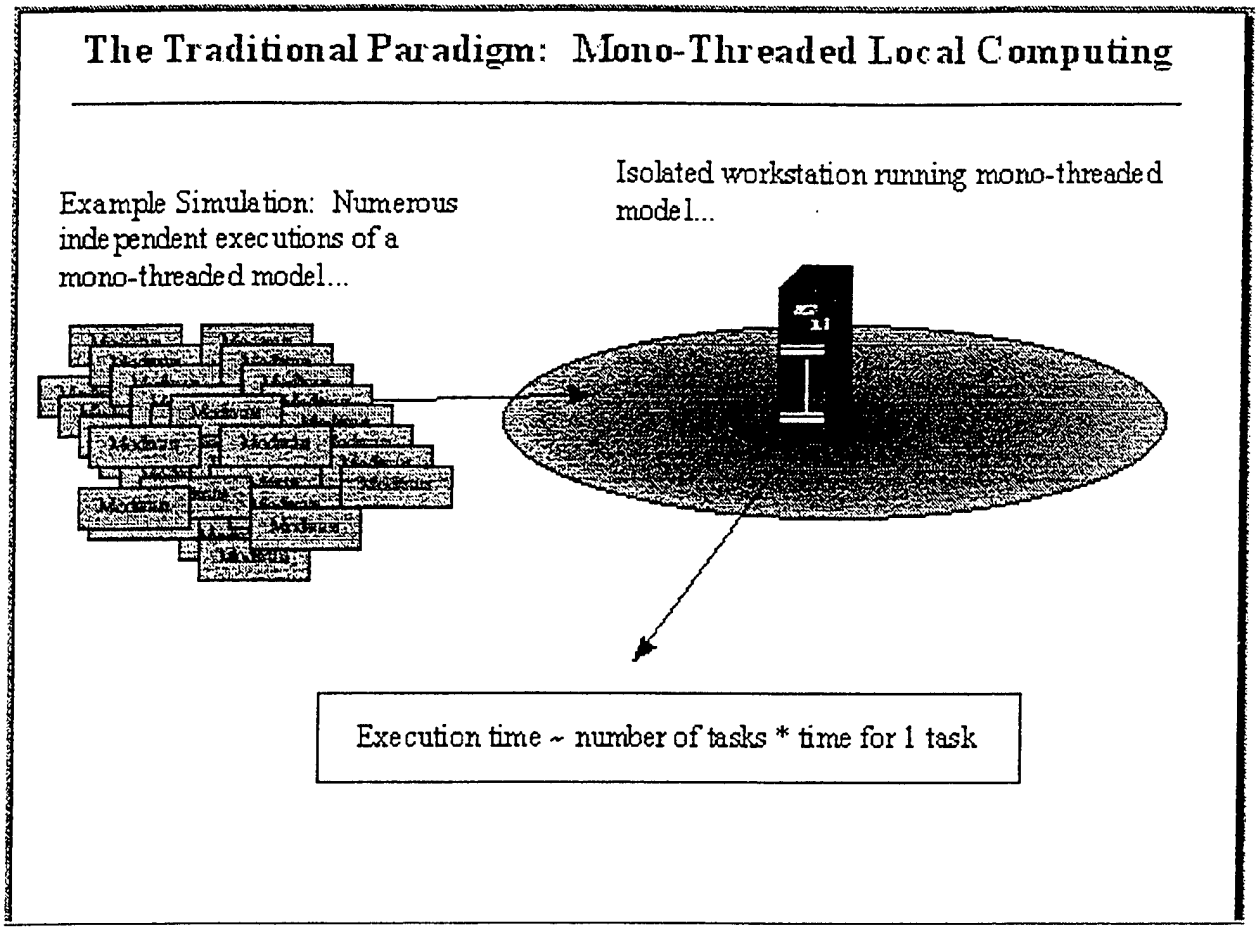


FIGURE C-2. THE TRADITIONAL COMPUTING PARADIGM. ISOLATED WORKSTATION RUNNING MULTIPLE INSTANCES OF A MODEL.

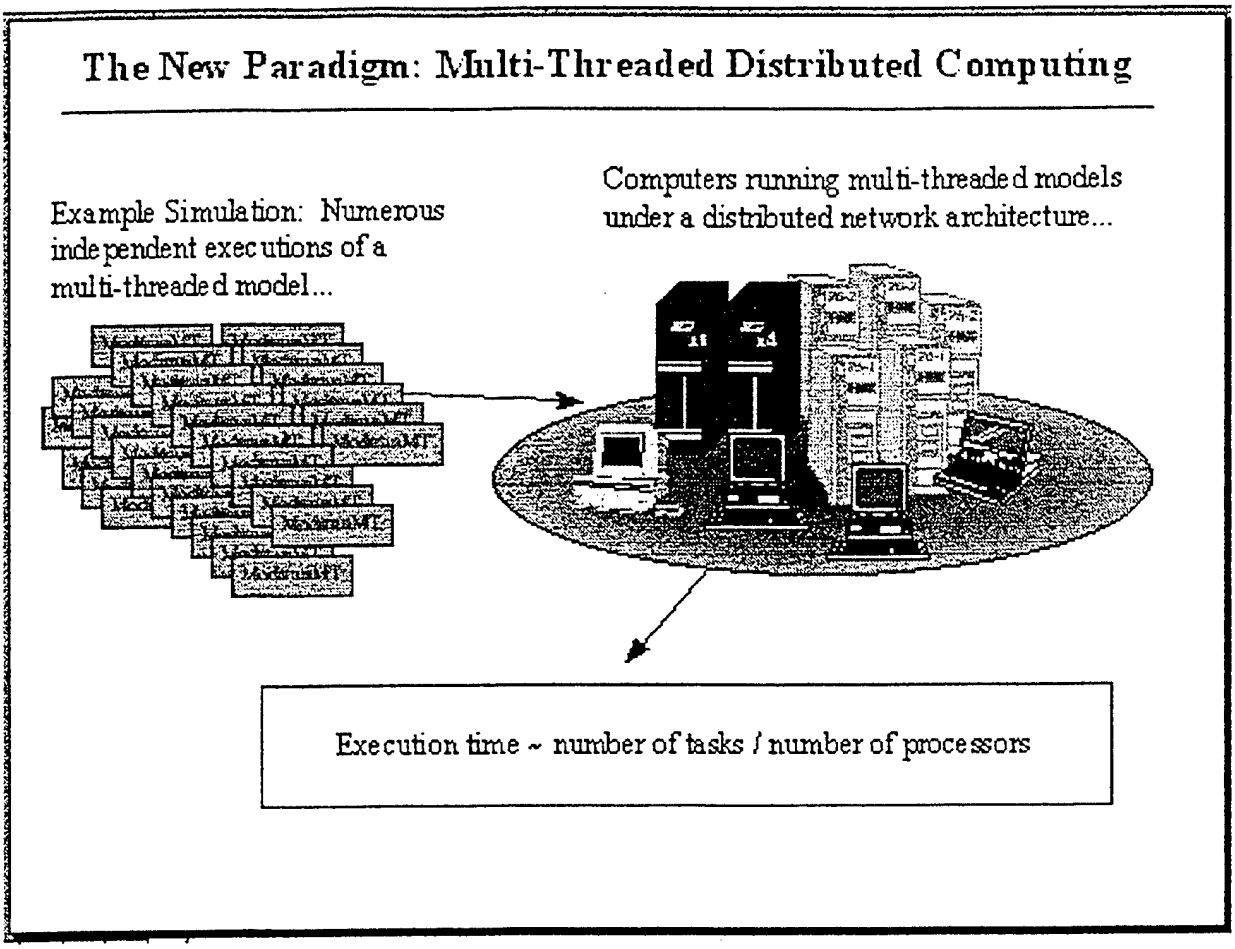


FIGURE C-3 A NEWER PARADIGM. MULTIPLE WORKSTATIONS COOPERATING TO IMPROVE COMPUTATION SPEED.